Carnegie Mellon

institute for SOFTWARE RESEARCH

Software Architecture

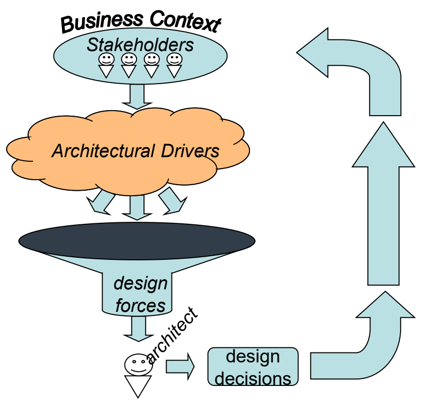
Architectural Structures

Instructor: Nguyễn Đức Mận
0904 235 945
mannd@duytan.edu.vn

Nội dung được xây dựng dựa trên tài liệu của Prof. Matthew Bass 2018 - ISR

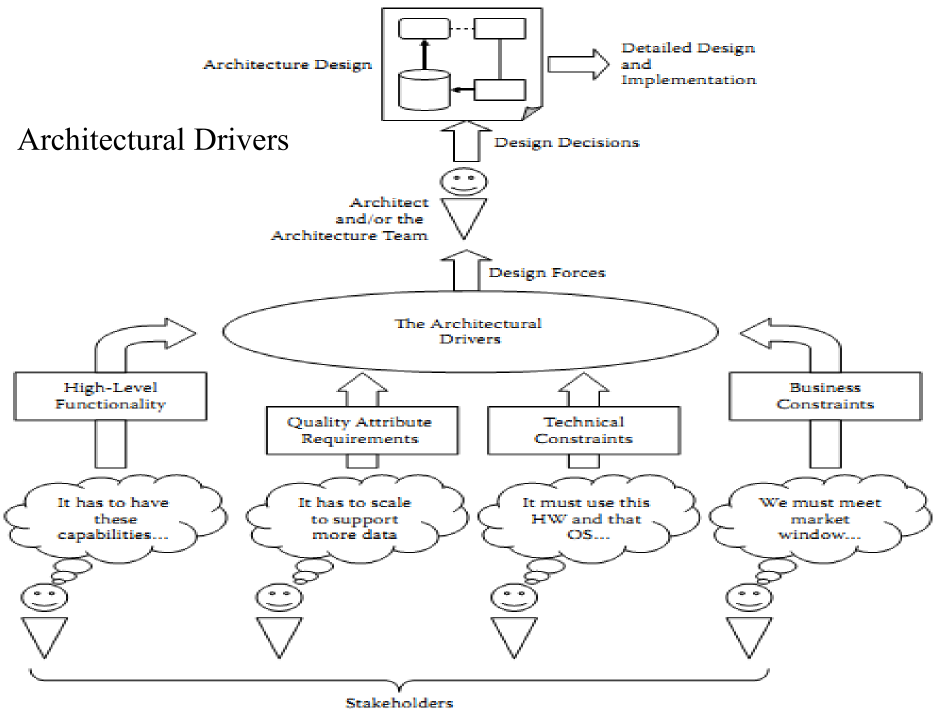
1

Back To The Goal



2

Architectural Drivers


Architecture Design
Detailed Design and Implementation
Design Decisions
Architect and/or the Architecture Team
Design Forces
The Architectural Drivers
High-Level Functionality
Quality Attribute Requirements
Technical Constraints
Business Constraints
It has to have these capabilities...
It has to scale to support more data
It must use this HW and that OS...
We must meet market window...
Stakeholders

3

Recall This Definition

- "The software architecture of a program or computing system is the structure or structures of the system, which comprise the software elements, the externally visible properties of those elements, and the relationships among them. *"

*Bass, L.; Clements, P. & Kazman, R. *Software Architecture in Practice, Second Edition*. Boston, MA: Addison-Wesley, 2003.

4

Who Motivates Design?

- Stakeholders motivate design decisions – they are anyone that has an interest in the software system and might include:
  - customers, users, markets
  - developers, designers, testers
  - project/product managers, marketers
  - maintainers, installers, trainers... and many others...
- Stakeholders are MORE THAN USERS!
- Stakeholders have different concerns that they want to guarantee and/or optimize
- Stakeholders are the source of *architectural drivers*

Types of Stakeholders – Users

- *Users* are the people who will use the system to do their job
  - Primary concern is a system's ease of use and utility with respect to getting their job done
- Not all users have the same concerns, needs, or expectations of the system
- Consider these two types users:
  - end users
  - system administrators

Types of Stakeholders – Users

- End users are often domain experts
  - Aircraft pilot, accountant, automobile driver, homemaker balancing the checkbook
  - They can be technologically challenged, but not always
- System administrators are concerned with a system's ease of use in terms of their ability to
  - configure the system
  - manage users
  - establish security and detect security breaches
  - back up information
  - recover and rebuild the system
- These two users have very different expectations

Types of Stakeholders – Customers

- *Customers* are the people who pay for the system's procurement. Note that customers are not always users
- Customers' concerns include the system's
  - cost
  - functionality
  - lifetime
  - development time/time to market
  - quality
  - flexibility to do many things on delivery da and over its lifetime

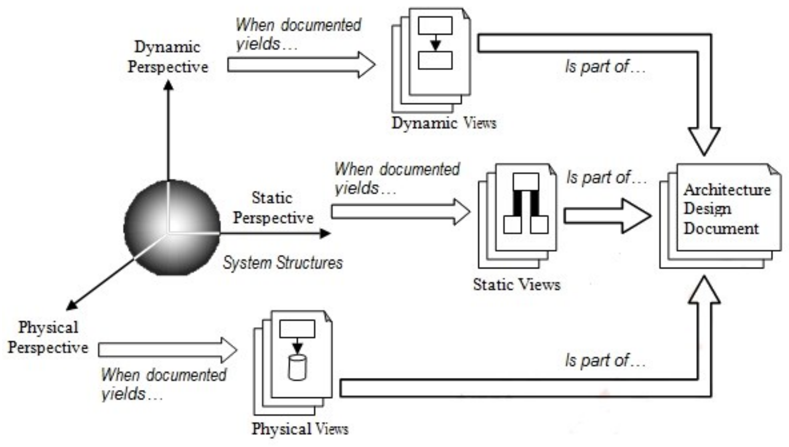
Types of Stakeholders – 4

- *Management* stakeholders include those managers from the
  - development organization
  - customer organization
- Managements' concerns include
  - amortizing development costs
  - maintaining the workforce's core compet and organizational training
  - investing to achieve strategic goals
  - keeping development costs as low as po
  - adhering to the development schedule
  - maintaining product quality

Types of Stakeholders – 5

- *Developers* are concerned about languages, technology, and the best mix to solve the problem
- *Maintainers* want a system they can fix, improve, modify, extend, and so forth
- *Administrators* want a system they can tune, configure, deploy, and so forth
- *Marketers* want features that meet or exceed those of the competition at a competitive price

Put all together



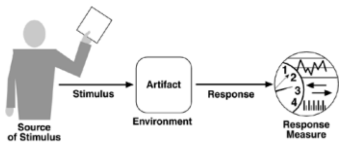
Perspective, Structures, Relationships, and Views

More on this later when we discuss documentation

Perspective	Example Structures	Example Relationships	Views
Dynamic	Processes Treads :	Dataflow Events :	Component and Connector
Static	Layers Code Modules :	Depends Uses :	Module
Physical	Computers Sensors :	Serial Line Wireless :	Allocation

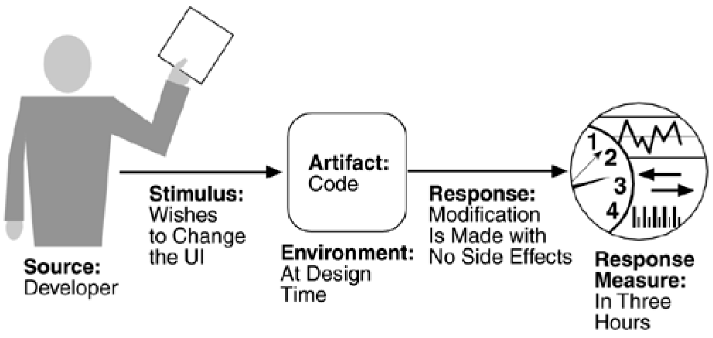
Specifying Quality Attribute Requirements

- We use a common form to specify all quality attribute requirements as scenarios.
- The form has six parts:
  1. **Stimulus**
  2. **Stimulus source**
  3. **Response**
  4. **Response measure**
  5. **Environment**
  6. **Artifact**



Review?

- "A developer wishes to change the user interface to make a screen's background color blue. This change will be made to the code at design time. It will take less than three hours to make and test the change and no side effect changes will occur in the behavior"
  - What QA do you mention about?

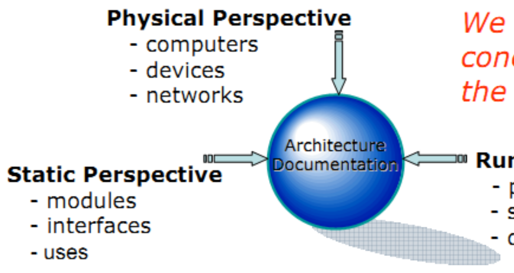

Source: Developer
Stimulus: Wishes to Change the UI
Artifact: Code
Environment: At Design Time
Response: Modification Is Made with No Side Effects
Response Measure: In Three Hours

Sample Concrete Performance Scenario

- Users initiate transactions under normal operations. The system processes the transactions with an average latency of two seconds.
  1.
  2.
  3.
  4.
  5.
  6.

What Is Structure?

- A representation of a structure is usually called a view of the system
- Software architecture documentation is comprised of a collection of views



We will revisit this concept throughout the course…

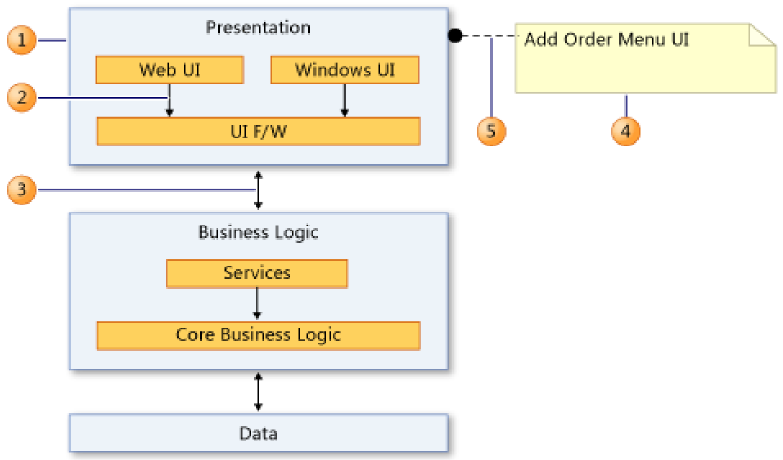
Structures and Views

- A *view* is a representation of a coherent set of architectural elements.
  - A set of elements and the relations among them.
- A *structure* is the set of elements itself, as they exist in software or hardware.
- In short, a view is a representation of a structure.
  - For example, a module *structure* is the set of the system's modules and their organization.
  - A module *view* is the representation of that structure
    - Documented according to a template in a chosen notation, and used by some system stakeholders.
- Architects design structures. They document views of those structures.

17

Some Useful Module Structures

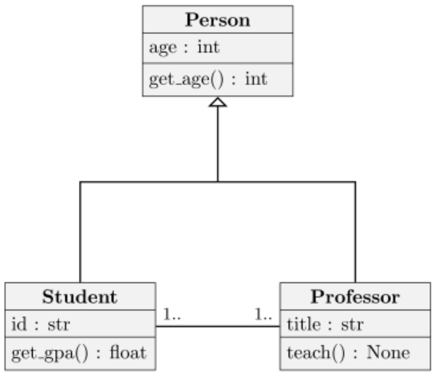
- Layer structure



18

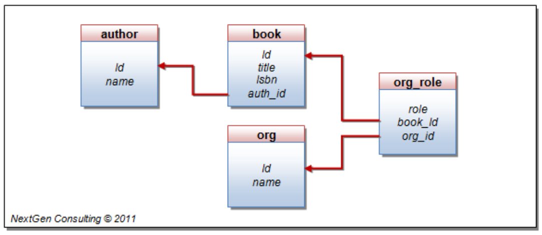
Some Useful Module Structures

- Class (or generalization) structure
  - The module are classes.
  - The relation is inherits from or is an instance of.
  - Allows reasoning about reuse and the incremental addition of functionality.
  - Object-oriented analysis and design approach uses this structure.



Some Useful Module Structures

- Data model
  - The data model describes the static information structure in terms of data entities and their relationships.
    - Tables, Attributes

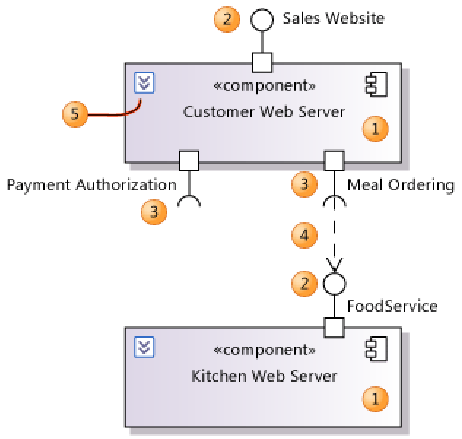


Some Useful C&C Structures

- The relation is attachment
  - Showing how the components and the connectors are hooked together.
  - The connectors can be familiar constructs such as "invokes."

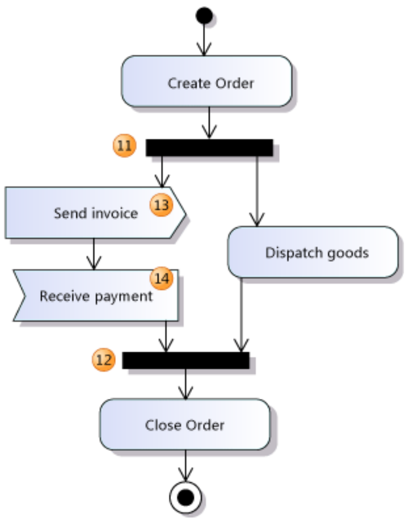
Some Useful C&C Structures

- Service structure
  - The units are services
    - interoperate by eg. SOAP.
  - Allows using external components or services



Some Useful C&C Structures

- Concurrency structure
  - The units are components
  - The connectors are their communication mechanisms.
  - The components are arranged into logical threads.
  - Helps find opportunities for parallelism or possible resource contention.



Architectural Alignment

- Now that we have the criteria defined we need to make appropriate design decisions
- In order for this to happen we need to:
  - Understand the relationship between design decisions and systemic properties
- This is similar to what happens in other engineering disciplines
  - Think the "Engineering Handbook" for mechanical engineering

Typical Design Approaches

- What are the typical strategies for making design decisions?
  - Yourdon's structured design
  - Object oriented approaches
- Attempts to routinize the design process
- Typically perform some kind of functional decomposition of the system
  - We may use heuristics such as coupling and cohesion

Results of Functional Decomposition

- We don't explicitly consider the impact on systemic properties
- Thus we don't know what properties will be promoted and inhibited until we have built the system
  - This is why performance engineers are typically in "fire fighting" mode
- We can only do so much by "tweaking" the system

What Should We Do?

- We should explicitly consider the structures that support the desired properties
- For example – if you a wide span that can't be supported from below you would consider a suspension bridge
  - However, you would know that it has allot of flex (thus wind shear can be an issue)

Software

- We can do the same with software
- Certain kinds of software structures support modifiability
- Others are very predictable from a performance perspective
  - And so on …

Structure Influences Systemic Properties

- The structure of the software impacts the properties the system will exhibit
- Different kinds of structures impact different properties e.g.
  - The structure of the code itself typically impacts the modifiability of the system
  - Performance is most often impacted by the run time structures
- Once we understand the need (as discussed in the previous section) we begin to think about structures and the properties they promote

Software Structures

- What kinds of structures do we have?
- In software we can generally think of categories of structures
  - Static structures
  - Dynamic structures

Static Structures

- These are the software elements that exist prior to compilation and execution
- That means that static elements are … code
  - Code can be organized and structured in a variety of ways, but it is all code
  - And they are related in all the ways in which code can be related
- Static elements can be called many things including:
  - Modules
  - Subsystems
  - Layers

Dynamic Structures

- These are runtime elements (elements that exist during execution)
- These are units of execution and could be:
  - Threads
  - Processes
  - “Components” – think Session or Entity Beans in EJB
  - Shared data
  - “Logical threads” – such as in .Net
- Relationships at the highest level are things like:
  - Messages
  - Control flow
  - Data flow

Composition of Structures

- The selection and arrangement of structures dictates the systemic properties supported by the system
- For example:
  - If we expect to change a particular aspect of the system the structure of the overall code dictates how much effort it will require
  - Think about changing some aspect of the UI
  - The extent to which the business logic is embedded in the UI will impact effort
- The same goes for other properties
- Over time we've learned what ***patterns*** of structures support particular properties

What is a Pattern? – 1

- When experts work on a particular problem it is unusual for them to invent a totally new solution - *They recall a similar problem and reuse the essence of the solution.*
- This expert behavior is called thinking in *problem-solution* pairs and is common in many domains such as architecture, economics, and software engineering.
- Patterns are common in software architecture designs and are of finer grained detail than styles.

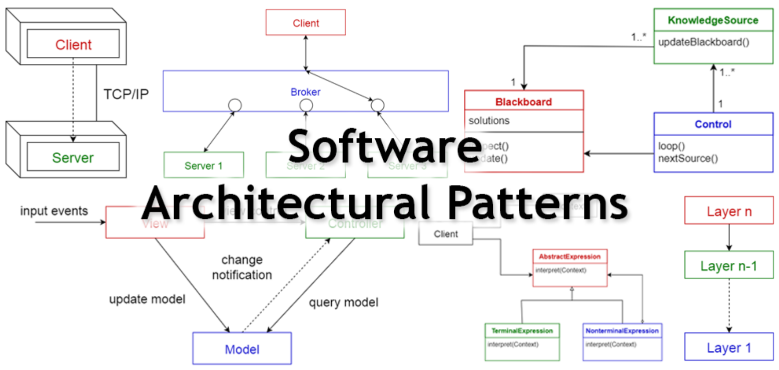
What is a Pattern? – 2

- Patterns come from the practitioners not academia
  - Codified patterns comes from identifying and abstracting specific problem-solution pairs and distilling the common features found in practice.
  - An understanding of patterns can help you build systems based on the collective experience of skilled software engineers.
  - Architectural patterns are not architectures but they are useful steps in creating an architecture.

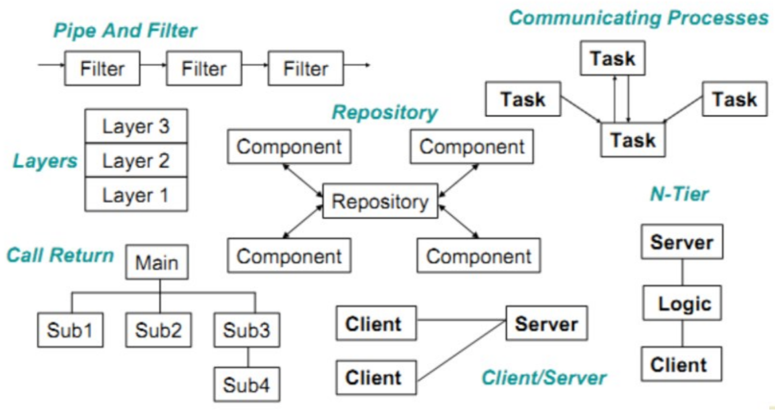
Architectural Patterns – 1

- A software *architectural pattern* is a like a fundamental or primitive structure
- This is similar to fundamental structures found in building architecture: arch or vault, lintel, truss, and so forth.
  - Patterns define families of general architectural structures such as client-server, pipe-and-filter, and so forth.
  - Patterns possess known properties.
  - The selection of an architectural pattern is often the architect's first major design choice.

Architectural Patterns – 1



Architectural Patterns – 2



Message Broker

- https://www.oreilly.com/library/view/pattern-oriented-software-architecture/9781119963998/chap12-sec005.html
- https://viblo.asia/p/message-broker-la-gi-so-luoc-ve-rabbitmq-va-ung-dung-demo-djeZ1PVJKWz
- https://edwardthienhoang.wordpress.com/2020/05/04/system-design-co-ban-phan-16-message-broker/
- https://topdev.vn/blog/kafka-la-gi/
- https://techinsight.com.vn/kien-truc-message-queue-trong-microservice/

Architectural Patterns – 2

- Architectural patterns define
  - a set of elements and responsibilities assigned to the elements
  - a set of relationships between the elements
  - a topological arrangement of the elements and relations
  - semantic rules for how the elements and relations connect, interact, and transform
- If these are adhered to, a particular pattern will promote some properties and inhibit others.

Architectural Patterns – 3

- It can be helpful to initially examine patterns in their idyllic or pure forms, however
  - pure patterns are rarely found in practice
  - systems in practice
    - regularly deviate from pure forms
    - typically combine many architectural patterns simultaneously
- Architects must understand patterns in their pure forms to understand their strengths, weaknesses and the consequences of deviating from them.

A Short Catalogue of Patterns

- There is a not complete, comprehensive, commonly agreed to list of architectural patterns. Here are a few:
  - communicating processes, distributed processes
  - event oriented systems (both implicit and explicit)
  - batch sequential, pipes and filters, process control
  - repository, blackboard
  - interpreter, rule-based systems
  - layers
  - object oriented, main program functional call

Interpreting Patterns – 1

- There is no complete, standard set of patterns.
  - We have few recognized architectural structures that are well codified.
  - Codification of the structures and the underlying guiding principles for their use is often incomplete, informal, and imprecise.
  - When a structure is documented, we (the architect) are often left with the details of sorting out the granularity of design that is addressed by the structure and deciding if the structure is applicable for our needs.

Interpreting Patterns – 2

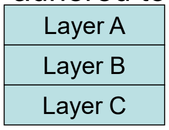
- As you read about patterns and/or discover and document your own, use the following as a framework to analyze them and guide you in their use. When confronted with a pattern, think in terms of
  - perspective and view of the representation
  - topological arrangement (usually the canonical picture or representation)
  - elements and relations (fundamental structures)
  - semantics of the pattern
  - qualities promoted by the pattern
  - qualities inhibited by the pattern

Example: Layers – 1

- The layered pattern is used to structure applications that can be decomposed into groups of elements in each of which address a different level of abstraction.
- Elements
  - elements are layers and are typically libraries or collections of related services
  - specific and related functional responsibilities are assigned to each layer
- Relations
  - relationships between layers is often implicit and is usually call-return oriented

Example: Layers – 2

- Perspective and view
  - code oriented; usually module view, although in practice mixed perspectives are often and incorrectly utilized
- Topology
  - layers are positions adjacent to one another
  - relationships between layers are not usually explicitly depicted – this is OK, if the pattern semantics are adhered to.

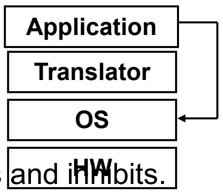


Example: Layers – 3

- Semantics
  - Layers can only call on services of the next lower layer; that is, each layer provides services to the layer above it and calls on services provided by the layer below it.
- Example properties promoted
  - modifiability, portability, reusability
- Example properties inhibited
  - performance, size of implementation; that is executable code built using this pattern will be larger

Why are the Semantics Important?

Let's consider an example…



This structure has semantics, properties it promotes and inhibits. Knowing these enables early analysis of the properties.

Assume that we chose this structure *without any analysis* and we find out that the system is too slow,… what can we do?

We can bridge layers,… but what happens to portability?

These decisions are only good or bad depending upon the relative importance of portability and performance – *but without the proper expertise, early analysis is impossible!*

Importance of Architectural Patterns I

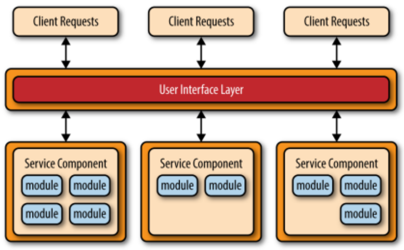
- Architectural patterns
  - address reoccurring design problems that arise in specific design situations and provide a general solution to it
  - document existing, proven design experience – *they facilitate knowledge reuse*
  - identify and specify abstractions that are above the level of detailed design abstractions
- Patterns provide a starting point for system designers.
- Patterns promote and inhibit various properties in known ways.

Importance of Architectural Patterns II

- Patterns provide a common vocabulary for designers.
  - Serves as a means for describing designs to various problems and discussing their merits and shortfalls.
- Architectural designs are typically composed of patterns, *or ensemble of patterns*.
  - Knowing something about patterns can help architects reason about the properties of an ensemble.

Importance of Architectural Patterns III

- An understanding of architectural patterns provides leverage for
  - Designers: those architects trying to select structures to meet functional and quality attribute properties
  - Evaluators: those who study architecture designs to ensure that they meet the functional and quality attribute requirements



Tactics

- A *tactic* provides a scheme for refining the elements of a software system and the relationships between them.
- Tactics
  - are independent of a particular programming language
  - consist of finer granularity details than architectural patterns
  - directly promote quality attribute properties
  - can be used to refine coarse grained design patterns

Example – 1

- In practice, forces act upon structures and important design considerations include the magnitude and frequency of occurrence, distribution, and nature as static or dynamic
  - Such forces cause stresses, deformations and displacements in structures.
  - Excessive forces can cause structural failure.
  - Assessment of the effects of forces on structures is the essential task of designers who can select various tactics to counteract or control their effects upon structures.

Example – 2

- Consider the effect of live loads such as snow, wind, or an earthquake, may have upon our initial structural choice of a vault.
  - To counteract these forces a mastic joint is a tactic that can be applied to preserve the *design intent* and will minimize the effect of the forces.
  - Note that mastic joint is not particular to the structural choice, but complements the structure to control a negative response.
  - So it is in software intensive system design…

Tactics – 1

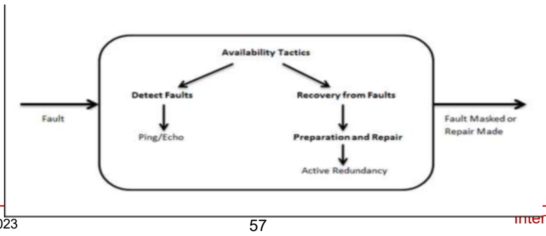
- Often the type of problem or domain will suggest a particular pattern.
- Once the architect has selected a pattern or ensemble of patterns that defines the overall structure of the system, many choices still remain.
- Tactics provide finer grained structural details that allow more control over quality attribute responses.

Tactics – 2

- Tactics are pre-packaged design options for the architect.
  - For example, to promote availability, we might chose redundancy as a tactic.
  - Tactics can be further refined by other tactics – *redundancy could be refined to data and/or computational redundancy tactics.*
- We will examine a few tactics in detail
  - availability
  - modifiability
  - performance
  - security

Availability Tactics – 1

- Fault detection
  - ping/echo: when one component issues a ping and expects to receive an echo within a predefined time from another component
  - heartbeat: when one component issues a message periodically while another listens for it
  - exceptions: using exception mechanisms to raise faults when an error occurs



Availability Tactics – 2

- Fault recovery
  - voting: when processes take equivalent input and compute output values that are sent to a voter
  - active redundancy: using redundant components to respond to events in parallel
  - passive redundancy: a primary component responds to events and informs standby components of the state updates they must make
  - spare: a standby computing platform is configured to replace failed components

Availability Tactics – 3

- Fault recovery and reintroduction
  - shadow operation: a previously failed component may be run in “shadow mode” before it is returned to service
  - state re-synchronization: saving state periodically and then using it to re-synchronize failed components
  - checkpoint/rollback: recording a consistent state that is created periodically or in response to specific events

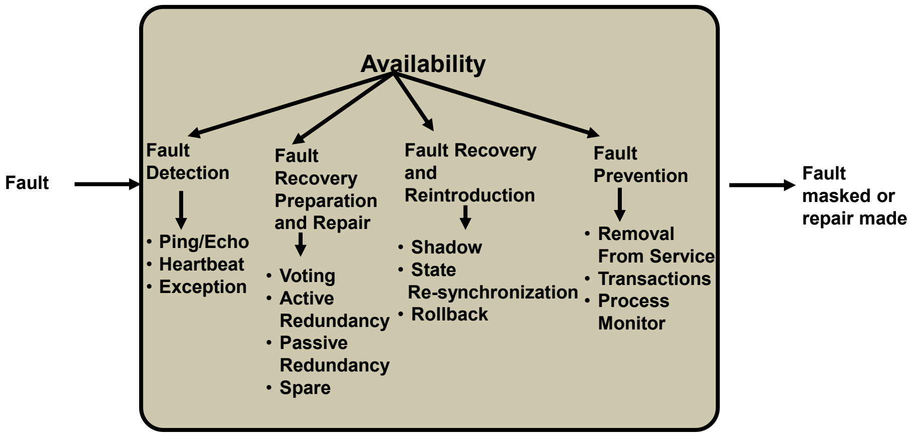
Availability Tactics – 4

- Fault prevention
  - removal from service: removing a system component from operation to undergo some activities to prevent anticipated failures (e.g., rebooting a component that will prevent memory leaks from causing failure)
  - transactions: the bundling of several sequential steps such that the entire bundle can be undone at once
    - prevents data from being affected if one step in a process fails
    - prevents collisions among several simultaneous threads from accessing the same data.

Availability Tactics – 5

- Fault prevention (continued)
  - process monitor: monitoring processes are used to monitor critical components and remove them from service and re-instantiate new processes in their place

Summary of Availability Tactics



Modifiability Tactics – 1

- Localization of modifications
  - maintain semantic coherence: ensuring that all the responsibilities in a module work together without excessive reliance on other modules
  - anticipate expected changes: designing and building a system with a set of envisioned changes in mind
  - generalize modules: creating modules that are more general and allowing them to compute a broader range of functions based on input
  - limit possible options: restricting the possible variations and modifications to the modules of a system

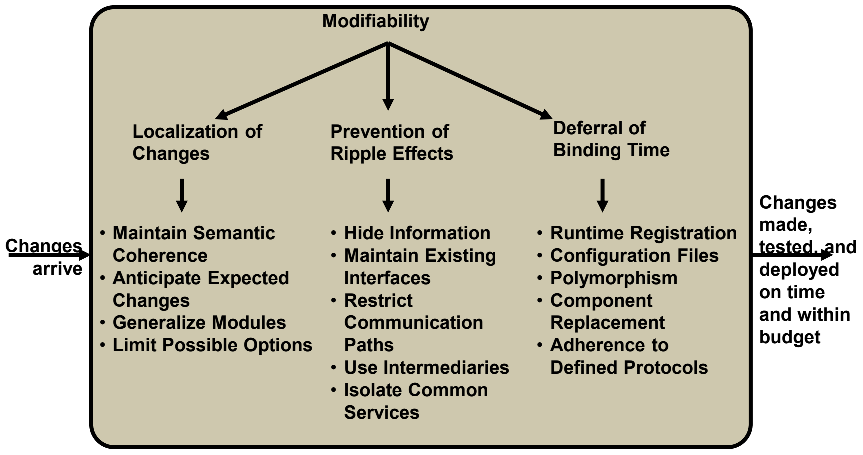
Modifiability Tactics – 2

- Prevention of accidental ripple effects
  - hide information: ensuring that all of a module's responsibilities are related and that it works without excessive reliance on other modules
  - maintain existing interfaces: adding additional interfaces, using adapters and stubs
  - restrict communications paths: restricting the modules with which a given module shares data
  - use intermediaries: using repositories, bridges, proxies, name servers, resource brokers, and so forth
  - isolate common services: providing common services through specialized modules

Modifiability Tactics – 3

- Deferral of binding time
  - runtime registration: supporting "plug and play" operation
  - configuration files: setting parameters and configuring elements at start-up/initialization time
  - polymorphism: allowing the late binding of method calls
  - component replacement: allowing load-time binding
  - adherence to defined protocols: allowing the runtime binding of independent processes

Summary of Modifiability Tactics


Modifiability
Changes arrive
Localization of Changes
• Maintain Semantic Coherence
• Anticipate Expected Changes
• Generalize Modules
• Limit Possible Options
Prevention of Ripple Effects
• Hide Information
• Maintain Existing Interfaces
• Restrict Communication Paths
• Use Intermediaries
• Isolate Common Services
Deferral of Binding Time
• Runtime Registration
• Configuration Files
• Polymorphism
• Component Replacement
• Adherence to Defined Protocols
Changes made, tested, and deployed on time and within budget

Performance Tactics – 1

- Resource Demand
  - increase computational efficiency: improving the algorithms used in critical areas will decrease latency
  - reduce computational overhead: reducing the need for and use of resources, thereby reducing the processing required
  - manage event rate: reducing the sampling frequency at which variables are monitored
  - control frequency of sampling: queuing the arrival of events and sampling them at a lower frequency

Performance Tactics – 2

- Resource management
  - introduce concurrency: processing requests in parallel, thereby reducing the time that tasks are blocked
  - maintain multiple copies of data and computations: reducing the contention that would occur if all data were accessed at a single location or computations were performed on a single resource
  - increase available resources: using more and/or faster processors, more and/or faster memory, more and/or faster networks, and so forth

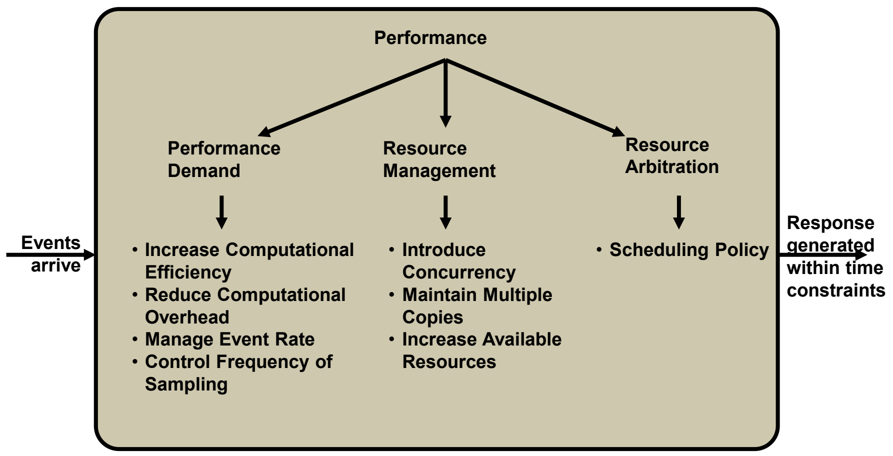
Performance Tactics – 3

- Resource arbitration
  - scheduling: analyzing the usage characteristics of each resource and choosing scheduling strategies that are compatible with the resources
    - first in, first out: Treat all requests as equals.
    - fixed priorities: Assign resources in a fixed order of priority.
    - dynamic priorities: Reorder priorities (e.g., round-robin) or schedule those resources with the earliest deadlines.
    - static scheduling: Preemption and the sequence of priority assignments are determined offline (compile time).

Discussion in Group

- Group discussion and propose tactics for improving the efficiency/Performance of banking transaction office at the Dien Bien Phu Branch?

Summary of Performance Tactics


Performance
Performance Demand
Resource Management
Resource Arbitration
Events arrive
• Increase Computational Efficiency
• Reduce Computational Overhead
• Manage Event Rate
• Control Frequency of Sampling
• Introduce Concurrency
• Maintain Multiple Copies
• Increase Available Resources
• Scheduling Policy
Response generated within time constraints

Security Tactics – 1

- Resisting attacks
  - authenticate users: ensuring identities by using passwords, digital certificates, and/or biometrics
  - authorize users: accessing control to data and/or services for authenticated users through user classes, groups, or roles
  - maintain data confidentiality: encrypting data and communication links
  - maintain integrity: through the use of encoded redundancy information such as checksums and hash results

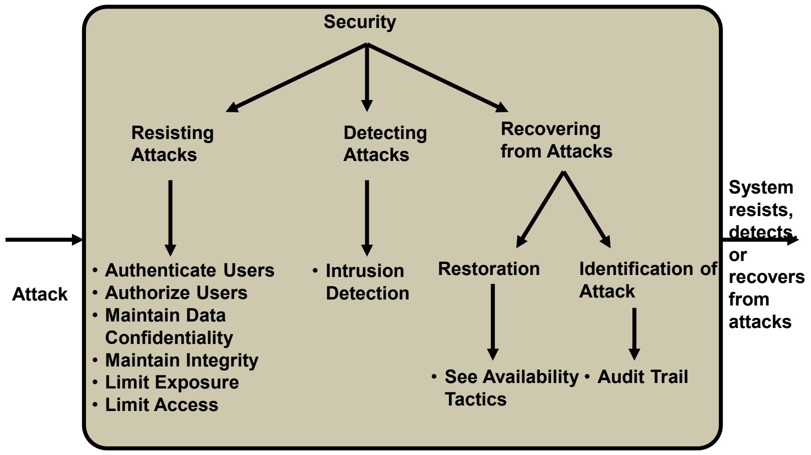
Security Tactics – 2

- Resisting attacks (continued)
  - limit exposure: allocating services to hosts and limiting the availability of services to a small number of hosts
  - limit access: using firewalls and demilitarized zones (DMZs)
- Detecting attacks
  - intrusion detections: comparing network traffic to historic patterns of use; recording and reporting any anomalies

Security Tactics – 3

- Recovering from an attack
  - restoring state: maintaining redundant copies of system administration data
  - identification of attackers: maintaining an audit trail of each transaction applied to data and access to services
  - see availability tactics

Summary of Security Tactics



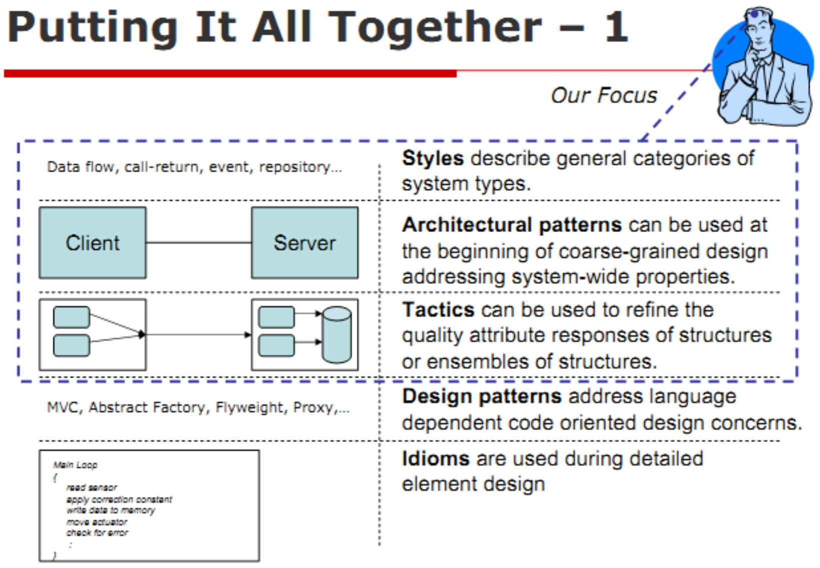
Design Patterns

- The term “design patterns” was coined by Gamma et. al.†
- The focus of Gamma’s work is on design patterns that are more fine grained than architectural patterns
  - Often they are decidedly language dependent (e.g. SmallTalk is basis language).
  - The term “object oriented architectures” is used.
  - The focus is code/module structure, not system structure.
  - Broad system properties and software intensive system design is not the focus.

Design Patterns

- It might be helpful to think of design patterns as abstractions addressing more detailed designs of interior structures.
  - designers of these structures are concerned with Balance, Focus, Harmony, Proportion, Rhythm, Scale, Subtlety, and their design principles and methods address these concerns

Putting It All Together – 1


Our Focus
Data flow, call-return, event, repository...
Client
Server
MVC, Abstract Factory, Flyweight, Proxy,...
Main Loop
{
read sensor
apply correction constant
write data to memory
move actuator
check for error
:
}

Styles describe general categories of system types.

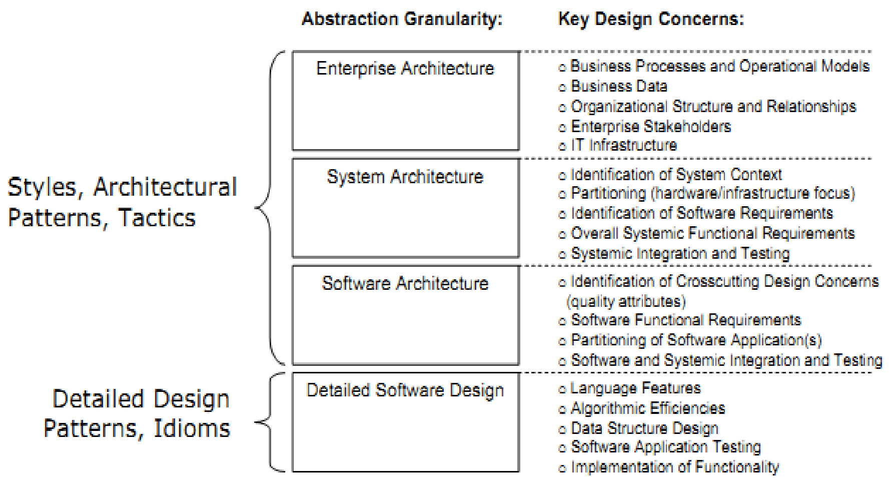
Architectural patterns can be used at the beginning of coarse-grained design addressing system-wide properties.

Tactics can be used to refine the quality attribute responses of structures or ensembles of structures.

Design patterns address language dependent code oriented design concerns.

Idioms are used during detailed element design

Putting It All Together – 2


Abstraction Granularity:
Key Design Concerns:
Styles, Architectural Patterns, Tactics
Detailed Design Patterns, Idioms
Enterprise Architecture
System Architecture
Software Architecture
Detailed Software Design
○ Business Processes and Operational Models
○ Business Data
○ Organizational Structure and Relationships
○ Enterprise Stakeholders
○ IT Infrastructure
○ Identification of System Context
○ Partitioning (hardware/infrastructure focus)
○ Identification of Software Requirements
○ Overall Systemic Functional Requirements
○ Systemic Integration and Testing
○ Identification of Crosscutting Design Concerns (quality attributes)
○ Software Functional Requirements
○ Partitioning of Software Application(s)
○ Software and Systemic Integration and Testing
○ Language Features
○ Algorithmic Efficiencies
○ Data Structure Design
○ Software Application Testing
○ Implementation of Functionality

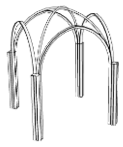
Putting It All Together – 2

- Each set of structures addresses different design concerns. Each choice of structures constrains the downstream design choices.

style

choice: gothic

architectural patterns



choice: vault

design patterns and idioms

choices: orientation, adornment, construction materials and techniques

So it is with software intensive systems

Domain Specific Patterns – 1

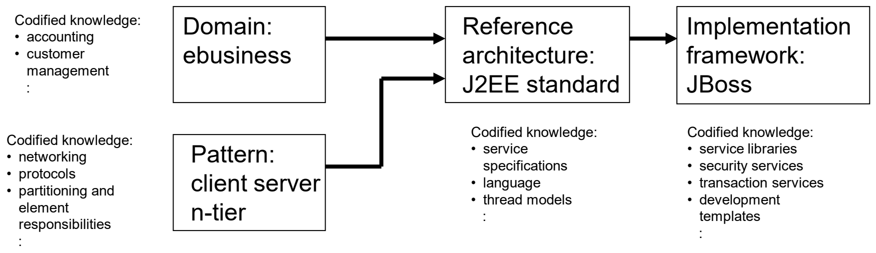
- It makes economical sense to codify architectural patterns that typifies a family of solutions to reoccurring problems unique to a particular domain.
  - promotes commonality among products
  - facilitates reuse and product lines
  - reduces learning curve
- Examples from industry include ERP/CRM, avionics, vehicle management systems, telecommunications, and many more.

Domain Specific Patterns – 2

- *Architectural patterns* can lead to *reference architectures* which are knowledge models.
- Reference architectures can be used to create to *implementation frameworks*.
- Implementation frameworks are not the product architecture but provide a significant economic advantage by reducing.
  - the amount of time spent design from scratch, coding and testing similar functionality
  - the number of defects introduced into the product

Domain Specific Patterns – 3

- Consider web oriented business applications.

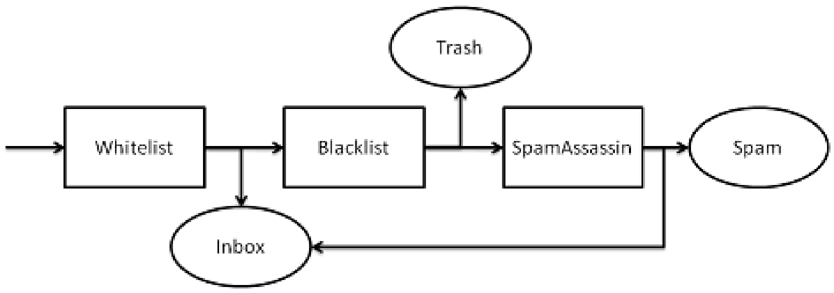


- While the JBoss implementation framework is not the complete product architecture, it takes us a long way toward the solution.

Exercise

- An e-mail system filters incoming e-mails with a whitelist (e-mails from senders on the hitelist are accepted), a blacklist (e-mails from senders on the blacklist are deleted), and the Spamassassin tool (e-mails that do not pass this check are marked as spam). The system will run on a single-core server machine, but may be moved to a multi-core server if the load gets too high.

Solution



Bài tập

- Mỗi nhóm thảo luận về đề tài Capstone
  - Mỗi nhóm đề tài thảo luận
    - QAs? Tactics gì?
    - Technical constraints
    - Business constraints

Summary – 1

- We started by revisiting structures, views, and perspectives.
  - architectures are more than pictures they are real structures manifested in the implemented system
  - views are pictures of the structures of an as-built or to-be-built system from a specific perspective
  - views are constrained by the types of elements and the relations between them
  - stakeholders perceive a system from different perspectives so a variety of views will be required for the architecture representation to serve as a communication vehicle

Summary – 2

- We discussed the essential qualities of an architectural pattern:
  - elements and relations (structures)
  - topology
  - perspective
  - semantics
  - quality attributes promoted and inhibited
- A case was made for
  - the importance and role of patterns in practice
  - exploiting patterns, reference architectures, implementation frameworks

Summary – 3

- We differentiated between styles, patterns, and tactics, and while there are no fixed rules, we discussed
  - architecture styles, patterns, and tactics and their roles in designing a software intensive system architecture
  - the general concept of tactics and how they are applied to achieve quality attributes

References

- Bass, L.; Clements, P. & Kazman, R. *Software Architecture in Practice.* Boston, MA: Addison-Wesley, 1998.
- Bass, L.; Clements, P. & Kazman, R. *Software Architecture in Practice, Second Edition.* Boston, MA: Addison-Wesley, 2003.
- Buschmann, F.; Meunier, R.; Rohnert, H.; Sommerlad, P.; Stal, M. *Pattern Oriented Software Architecture: A System of Patterns.* West Sussex England: John Wiley Ltd., 1996.
- Gamma, E.; Helm, R.; Johnson R.; Vlissides, J. *Design Patterns.* Reading, MA: Addison-Wesley, 1994.
- Lattanze, Anthony *Architecting Software Intensive Systems* Boca Raton, FL: Auerbach, 2009
- Video about **Reliability, Availability, Maintainability and Supportability** https://youtu.be/CmvKa26IdSc